SE320 H4 Answers

1. The benchmark harness is structured to first do an initial run without gathering performance statistics because the initial run takes more time due to cache loading, CPU, etc. and would inaccurately affect the statistics. The benchmark is then designed to do multiple runs (10 in total) and record the timing results in an array. Multiple runs are recorded as performance always varies, to gather an accurate understanding of performance statics are computed for the runs. It is also important to note that all values are hardcoded, and that no other processes are running concurrently which can affect performance as well.

2. insert/delete/lookup
    1. 90/10/0
    2. 10/10/80
    3. 0/0/100
3. 90/10/0
   Warmup: 229,823,916ns
   Min: 84071833ns
   Max: 96239000ns
   Avg: 96239000ns

   10/10/80
   Warmup: 111,057,125ns
   Min: 49558,958ns
   Max: 77662917ns
   Avg: 60404721ns

   0/0/100
   Warmup: 29,567,416ns
   Min: 6675083ns
   Max: 10084416ns
   Avg: 7162666ns

4. The 90/10/0 operational profile has a higher throughput. This operational profile consists of a majority inserts, I would expect the insertion operation to take the most time in a Red Black BST because after an

insertion, parent's colors might need to be changed in addition. Upon using VisualVM, I can confirm my hypothesis as insertion takes the most significant amount of time, even when it's a small percentage of operations.

5. The warm up times are longer by one significant figure! Typically a warmup time (the first run) takes longer as the program needs to load and process additional data, caches, initialize variables, etc.
6. The operation profile is very inaccurate because according do the benchmark code the gets keys from the BST according to what was put. In this operational profile nothing is put and therefore the size of BST is 0. In a true read-only application values would already be stored in the BST.
7.

c.

put()
0.066 ±(99.9%) 0.001 us/op [Average]
13.946 ±(99.9%) 0.177 ops/us [Average]

get()
0.007 ±(99.9%) 0.001 us/op [Average]
148.316 ±(99.9%) 0.381 ops/us [Average]

delete()
0.007 ±(99.9%) 0.001 us/op [Average]
147.046 ±(99.9%) 0.852 ops/us [Average]

d. I noticed this issue when I started to run tests, and saw that the delete benchmark test was run first because they benchmark tests are run in alphanumeric order. I tried @Order to force a test order but that didn't work, so I just put the char 'z' in front of the test that should not be run first. The first benchmark test that needs to be run is put, so values can be stored in the tree, then get should be implemented because the tree's values are now loaded, finally delete should be implemented after. If delete was implemented before get, the timings would be incomparable as the the tree size would be shrinking.